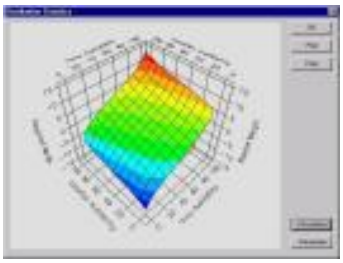


Programming Exercise 9:
Using Arrays

Purpose: Reinforce the use of arrays.
Background readings from textbook: Liang, Sections 7.1–7.2.

Due date for section 001: Monday, March 21 by 10 am
Due date for section 002: Wednesday, March 23 by 10 am

Overview
There are common operations that we typically use with arrays. Among these are loops to control "moving through an array" so that we can access each array element one at a time, sorting the array so that the values are in a proper sequence (increasing or decreasing order), and searching the array to locate a particular element or elements that fit a particular criteria. In the last lab, for instance, we searched for the elements less than the average. In this lab, we will explore some of these operations and gain further experience using arrays.

Part 1: Inputting an Array
If you recall in lab 8, we asked the user to input the number of elements that were to be placed into the array. This may not always be possible or convenient. The user may have hundreds of values to enter but doesn't want to count them all. Or, as we will cover later in the semester, the user may want to load the array values from a data file in which case the user doesn't know how many elements there will be. So, asking the user "how many elements are there" to both create the array (using `new type[number]`) and control a for loop to input into the array are not always a practical approach. Here, we look at how to use a while loop to input into an array. Consider the following.

```
int[] values=new int[100];
int number=0, temp;
Scanner in=new Scanner(System.in);
System.out.println("Enter your first value, negative to end  ");
temp=in.nextInt();
while(temp>=0)
{
      values[number]=temp;
      number++;
      System.out.println("Enter next value, negative to end  ");
      temp=in.nextInt();
}
```

We create an array of 100 ints where the variable `number` is the number of values currently in the array, initialized to 0, `temp` is the current value being input and `in` is our Scanner. We will assume the user is only going to enter positive or 0 values. We ask for the first value and then we reach our loop. As long as the value entered is not negative, we enter the loop body, insert it into values, increment number, and get the next value. This repeats until temp is negative. By the time the loop terminates, values is full of the numbers input and number is equal to the number of items input into the array. Notice if number is 10, there are 10 elements in the array, at indexes 0 through 9 (not 10!)

There is one flaw with this code. Notice how the array size is set to 100. Why 100? We don't know how many elements the user might enter, unlike in lab 8 so we pick a size. 100 seems large enough. What happens if the user enters 101 values? If number is 100 and we do values[number], this is the same as values[100]. This leads to an error known as an ArrayIndexOutOfBoundsException. This arises because we tried to use an index in our array that was out of range (recall a size of 100 means legal indices of 0 to 99). How can we prevent this from happening? We should modify our while loop to be `while(temp>=0&&number<100)`. In this way, we exit the while loop if the user enters a negative value *or* the variable number exceeds the size of the array, 100. Why did we use < 100 instead of <= 100? By using <= 100, the user could still enter a 101st value and then it would be placed at values[100] which causes the error noted above. Making this kind of mistake in your logic is common and is known as an "off-by-one error".

Part 2: Sorting and Searching an Array

Users will often enter data in a seemingly random way but want to view that data in some sorted fashion. For instance, if you were using a program to input a list of friend's names to later view that list alphabetized, you would probably not want to make the effort of sorting the list "by hand" before entering the data. So instead, we want to have the program sort it for us. Computer Science studies many sorting algorithms, each of which has different characteristics. For instance, the Bubble Sort is easy to implement but is not efficient. The Quick Sort can be very efficient but not always. The Merge Sort is usually slower than Quick Sort but never as inefficient as Quick Sort can be (we refer to algorithms efficiencies using terms like "best case", "average case" and "worst case" – Merge Sort has a better worst case than Quick Sort and both have a better worst and average case than Bubble Sort). We can also take into account memory utilization required by the sorting algorithm.

If you don't want to write your own sorting routine, you can use a built-in one. In Java, there is a class called `Arrays` which has a method called `sort`. To sort our earlier array, values, we can use `Arrays.sort(values,0,number);` In order to use Arrays.sort, you have to import the Arrays class, which is in java.util. Alternatively, you can write your own sort method as shown below. We would call this from our program using `sort(values, number);`

```
public void bubbleSort(int[ ] array, int n){
     int temp;
     for(int i=0;i<n-1;i++)   // loop for each item but last
          for(int j=0;j<n-1;j++)   // iterate through the loop
               if(array[j]>array[j+1])  // compare consecutive
               // elements and if out of order, swap them
               {
                    temp=array[j];
                    array[j]=array[j+1];
                    array[j+1]=temp;
               }    // end if
}    // end bubbleSort
```

Array: 12 6 9 3 4
First pass: 6 9 3 4 12
Second pass: 6 3 4 9 12
Third pass: 3 4 6 9 12
Fourth pass: 3 4 6 9 12

Notice that the array is already sorted after the third pass but the algorithm still makes 4 passes.

With an array sorted, we can search through the array for a particular item more efficiently. A simple search that looks at every item is known as a sequential search. Below, we see code that does this, looking for a particular item called `target`. If found, we remember where we found it in the variable `index`. Notice that even after the item is found, we continue to search through the array.

```
for(i=0;i<num;i++)
     if(names[i].equals(target)) index=i;
```

We can improve on this code by executing the loop once we have found the item. Imagine that the array stores 1 million elements. If target is found at index 21, we continue looking at the array for the remaining nearly 1 million elements. Very inefficient.

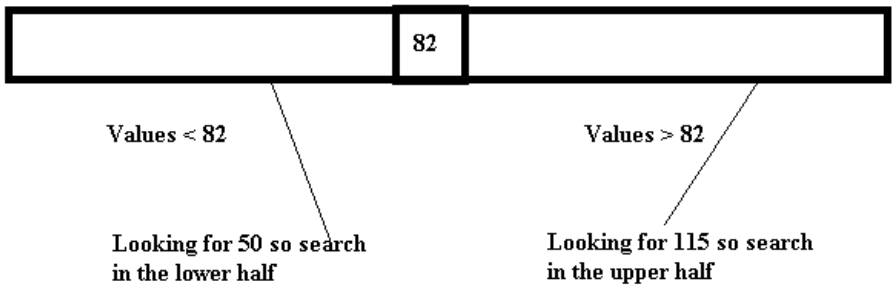
```
boolean found=false;
for(i=0;i<num&&!found;i++)
     if(names[i].equals(target)
     {
          index=i;
          found=true;
     }
```

Now, the loop terminates as soon as the name has been found. If there are a million names in the array, we won't necessarily search through all of them. We can improve on this even further. If the array is sorted and we find that names[i] is greater than target, it means that target is not to be found in the array. For instance, if we are looking for "Frank" and we come across names[50] which is "Fred", then we have passed "Frank" and can exit the loop. But an even better way to implement our search loop is to use a different search strategy. Consider looking for the word "magenta" in the dictionary. Are you going to start at page 1? No. Why not? Because you know that words that start with 'm' will not be at the beginning of the ordered list of words. More so, you would suspect words that start with 'm' will be somewhere near the middle. So you open the dictionary to the middle. Let's imagine that the page you open to starts with the word "purple". Then you would pick a new page further toward the front. This strategy works because the words are in sorted order. The strategy itself is called the binary search.

Using binary search, you would pick the midway point of the array and see if the item you are looking for is there. If not, you would either pick the midway point of the first half of the array (if the item you found is greater than the one you are looking for) or the second half of the array. In one comparison, you have then divided the space to search in half. See the figure on the next page. Now, if you are searching in the first half, look at the midway point of the first half and if you are searching in the second half, look at the midway point of the second half. Repeat until you find the item (or find that it's not there).

The binary search is much more effective than the sequential search. By continually dividing the search space in half with each comparison, we are reducing the amount to search by a factor of 2 (thus the name binary search). In sequential search, if there are 1000 items to search, in the worst case you have to look at all 1000 to find what you are looking for. With binary search, you only have to look

at 10 items (at most). If the array has 1 million items, sequential search may require looking at all 1 million but binary search only looks at 20 at most!



Part 3: Problem

We want to write a program which will input an array of ints from the user and perform various useful operations on the list:

1. Find the median value
2. Finding the average "step size" of the values
3. Find the most commonly occurring value

In order to accomplish the above three items, we will input the array, sort the array and print out the sorted array. We will then perform the 3 tasks above. Each of these tasks (input, sort, output, median, step size, most common value) will take place in separate methods all called from main.

In main, declare and instantiate an int array of 20 elements, and an int variable called `number` to store the number of elements in the array. Pass the array to an input method which will input, using a while loop, all of the items. This method will declare, use and close a Scanner. The while loop should iterate until either the user has input a 0 or has filled the array with 20 elements. The method should return the value in `number`. Since the method returns the number of elements, we need to assign number to the return value when calling the method in main using code like `number=getArray(array);` Main should then call the sort method (Arrays.sort(…);) and then the output method. To use the final three methods (common, median, step size, most common value), these will return a value that we want to output. You can either call the method from inside a println statement or use an assignment statement to store the returned value and then output it. For instance, if we have a method getMedian to output the median value, we might call it using either of the following.

```
System.out.println("The median value is "
      + getMedian(array, number);

median=getMedian(array, number);
System.out.println("The median value is " + median);
```

These three methods will return an int (#1, #2) or a double (#3). The median is the value in the middle of the array (e.g. array[number/2] once array is sorted). To find the step size, iterate through the array from 0 to n-2 and compute array[i+1]-array[i] and add this to a sum. When done, divide the sum by number-1. For instance, if our sorted array is 2, 3, 5, 9, 10, then the difference between each pair is 1, 2, 4, 1, which has a sum of 8, so when dividing by n-1 (4) we get an average step size of 2.0.

NOTE: if the array has fewer than 2 elements, this will not work. But in this assignment, we will assume that all inputs have at least 2 values.

To find the most common element in the array, we will use the fact that the array is sorted to simplify the task. Since the array is sorted, we need to count the number of consecutive items that have the same value. Once we reach a new value, we start recounting. We want to remember which item so far has occurred the most times and how many times. For instance, if our sorted array is 1, 1, 1, 2, 3, 3, 4, 4, 4, 4, 5, 5 and we are started at the first element, we count 3 consecutive 1s. When we reach 2, we start counting over. There is a single 2 so there are not more 2s than our most common so far. When we count 3s, we find two of them, again, not more than the 1s. We count four 4 which is more than three 1s, so we remember now four 4s. We need four variables for this, currentValue, currentCount, maxValue, maxCount. If array[i]==currentValue then we add 1 to currentCount. If not, then we have reached a new number. Now that we are done counting the number of the currentValue, we want to see if there were more of those than themaxCount. If currentCount is greater than maxCount, then we have a new maxValue and maxCount. A pseudocode description is given below.

Remember array[0] as currentValue and put 1 in currentCount
Set maxCount=0 and maxValue=0
Iterate from 1 to n-1 (we already looked at array[0], so start with array[1])
    If the new element == currentValue then add 1 to currentCount
    Otherwise // we have reached a new number
        If currentCount > maxCount then the most recent value occurred more
            often than the previous max so set maxCount = currentCount
            and maxValue = currentValue
        Since we have a new value, we have to reset curentCount to be 1 and
        currentValue to be a[i]
Return maxValue (not maxCount, we only want to know what value occurred the most, not how many times)

Write your program and run it on the following three data sets (one run per data set). Remember that you are limited to 20 inputs. Data set 3 goes over that limit. You should be prevented from entering the last thee values of that set. A useful thing to do is to output a warning should you reach the end of the array in this way.

Data set 1: 153, 100, 532, 100, 534, 154, 153, 100, 101, 153, 155, 153, 100, 193, 154, 153, 0
Data set 2: 66, 39, 38, 44, 65, 66, 67, 55, 44, 54, 66, 38, 67, 43, 66, 67, 33, 66, 51, 0
Data set 3: 10, 15, 19, 3, 7, 15, 4, 6, 19, 6, 15, 12, 9, 7, 15, 16, 10, 11, 8, 15, 21, 6, 15, 0

Part 4: Enhancement and Submission

Replace `Arrays.sort(array,0,number);` method call with `sort(array, number);` and write this method using the Bubble Sort code given in part 2. When done, submit your code and the outputs from the three data sets to the second and third runs (either as a separate text file or placed at the bottom of your source code in comments).